

# Evolving XSLT stylesheets

Néstor Zorzano, Daniel Merino, JLJ Laredo,  
JP Sevilla, Pablo García, JJ Merelo

February 2, 2008

## Abstract

This paper introduces a procedure based on genetic programming to evolve XSLT programs (usually called stylesheets or logicsheets). XSLT is a general purpose, document-oriented functional language, generally used to transform XML documents (or, in general, solve any problem that can be coded as an XML document). The proposed solution uses a tree representation for the stylesheets as well as diverse specific operators in order to obtain, in the studied cases and a reasonable time, a XSLT stylesheet that performs the transformation. Several types of representation have been compared, resulting in different performance and degree of success.

## 1 Introduction

Since the IT industry has settled in different XML (eXtensible Markup Language) [8] dialects as information exchange format, there is a business need for programs that transform from one format to another, extracting information or combining it in many possible ways; a typical example of this transformation could be the extraction of news headlines from a newspaper in Internet that uses XHTML<sup>1</sup>.

XSLT stylesheets (XML Stylesheet Language for Transformations) [6], also called *logicsheets* are designed for this purpose: applied to an XML document, they produce another. There are other possible solutions: programs written in any language that work with text as input and output, programs using regular expressions and SAX filters [14], that process each tag in a XML document in a different way, and do not need to load into memory the whole XML document. However, they need external languages to work, while XSLT is a part of the XML set of standards, and, in fact, XSLT logicsheets are XML documents, which can be integrated within an XML framework; that is why XSLT is, if not the most common, at least a quite usual way of transforming XML documents.

The amount of work needed for logicsheet creation is a problem that scales quadratically with the quantity of initial and final formats. For  $n$  input and

---

<sup>1</sup>An XML version of the Hypertext Markup Language (HTML) used in web pages.

$m$  output formats,  $n \times m$  transformations will be needed<sup>2</sup>. Considering that each conversion is a hand-written program and the initial and final formats can vary with certain frequency, any automation of the process means a considerable saving of effort on the part of the programmers.

The objective of this work is to find the XSLT logicsheet that, from one or several input XML documents, is able to obtain an output XML document that contain exclusively the information that is considered important from original XML documents. This information may be ordered in any possible way, possibly in an order different to the input document. This logicsheet will be evolved using Genetic Programming; XSLT programs are obviously not written in LISP (or Lisp-like s-expressions), as is usual in GP, but they also have a tree structure (as any XML document), since they are represented internally as a DOM (Document Object Model) tree.

In order to evolve XSLT logicsheets, we will have to take into account this structure. XML is an extensible markup language, in other words, a language that allows to define elements (tags) and the grammar that they follow. XML is based on the concept of encapsulation: document fragments are encapsulated in an area delimited by two tags. All XML documents have a tree structure (the so-called Document Object Model –DOM– tree) with a single root element that contains(encapsulates) all the contents of the document. In addition, XML elements have attributes which contain other information needed for the processing of the document. Sometimes, the elements and attributes have a syntax or semantics determined by Data Type Dictionary (DTD) or XSchema (equivalent concept that uses XML for its definition), in which case the document can be validated; however, in most applications what is called *well-formed* XML is more than enough.

Thus, XSLT provides a general mechanism for the association of patterns in the source XML document to the application of format rules to these elements, but in order to simplify the search space for the evolutionary algorithm, only three instructions of XSLT will be used in this work: **template**, which sets which XML fragment will be included when the element matching its **match** attribute is found; **apply-templates**, which is used to select the elements to which the transformation is going to be applied and delegate control to the corresponding **templates**; and **value-of**<sup>3</sup>, which simply includes the content of an XML document into the output file. This implies also a simplification of the general XML-to-XML transformation problem: we will just extract information from the original document, without adding new elements (tags) that did not exist in the original document. In fact, this makes the problem more similar to the creation of an *scraper*, or program that extracts information from legacy websites or documents. Thus, we intend this paper just as a proof of concept, whose generalization, if not straightforward, is at least possible.

We also take into account XPath [7] as a key element within the XML family of standards. XPath defines a way to locate a specific element within

---

<sup>2</sup>If an intermediate language is used, just  $n + m$ , but this increases the complexity of the transformation and decreases its speed.

<sup>3</sup>With **text** used for easy visualization of the final document

a XML document, by using references to specific nodes in the document, in an way similar to file access in a file-system tree; in the XPath specification, a document is considered as a tree, accessible by position. In addition, XPath provides a way to select groups of elements (*node-sets*) and to filter them by using predicates allowing, for instance, to select the element that occupies a certain position within a node-set.

The rest of the paper is structured as follows: the state of the art is presented in section 2. Section 3 describes the solution presented in this work. Experiments are described in section 4, with the automatic generation of XSLT stylesheets for two examples and finally the conclusions and possible lines of future work are presented in section 5.

## 2 State of the art

So far, very few papers about applying genetic programming techniques to the automatic generation of XSLT logicsheets have been published; one of these, by Scott Martens [11], presents a technique to find XSLT stylesheets that transform a XML file into HTML by using genetic programming. Martens works on simple XML documents, like the ones shown in its article, and uses the UNIX diff function as the basis for its fitness function. He concludes that genetic programming is useful to obtain solutions to simple examples of the problem, but it needs unreasonable execution times for complex examples and might not be a suitable method to solve this kind of problems. However, computing has changed a lot in the latest seven years, and the time for doing it is probably now, as we attempt to prove in this paper.

Unaware of this effort, and coming from a completely different field, Schmidt and Waltermann [12] approached the problem taking into account that XSLT is a functional language, and using functional language program generation techniques on it, in what they call *inductive synthesis*. First they create a non-recursive program, and then, by identifying recurrent parts, convert it into a recursive program; this is a generalization of the technique used to generate programs in other programming languages such as LISP [5, 13], and used thoroughly since the eighties [4].

A few other authors have approached the general problem of generating XML document transformations knowing the original and target structure of the documents, as represented by its DTD: Leinonen et al. [10, 9] have proposed semi-automatic generation of transformations for XML documents; user input is needed to define the label association. There are also freeware programs that perform transformations on documents from a XSchema to another one. However, they must know both XSchemata in advance, and are not able to accomplish general transformations on well formed XML documents from examples.

The automatic generation of XSLT logicsheets is also a super-set of the problem of generating *wrappers*, that is, programs that extract information from websites, such as the one described by Ben Miled et al. in [3]. In fact, HTML is

similar in structure to XML (and can actually be XML in the shape of XHTML), but these programs do not generate new data (new tags), but only extract information already existing in web sites. This is what applications such as X-Fetch Wrapper, developed by Republica<sup>4</sup>, do. The company that marketed it claims that it is able to perform transformation between any two XML formats from examples. Anyway, it is not so clear that transformations are that straightforward: according to a white paper found at their website, it uses a document transformation language.

### 3 Methodology

XSLT stylesheets have been inserted into tree structures, making them evolve by using variation operators. Each XSLT stylesheet is evaluated using a fitness function that shows the adjustment rate between generated XML and output XML associated to the example. The solution has been programmed using JEO [1], an evolutionary algorithm library developed at University of Granada as part of the DREAM project [2], which is available at <http://www.dr-ea-m.org> together with the rest of the project.

The generated XML documents are encapsulated within an XML tag whose name equals the root element from the input XML. Next, structures used for evolution and operators applied to them are described. These operators work on data structures and XPath queries within them.

The search space over possible stylesheets is exceedingly large. In addition, language grammar must be considered in order to avoid syntactically wrong stylesheet generation. Due to this, transformations are applied to predetermined stylesheet structures which have been selected. These transformations alter the structure and preserve the syntax. This limits search space, generating suboptimal solutions, so three stylesheets structures that are not changed by transformations have been selected.

Next we will describe the three different XSLT stylesheet structures that have been used in the experiments, and the operators that are applied on them.

#### 3.1 First structure

- The XSLT logic sheet will have three levels of depth. First level is the root element `<xsl:stylesheet>` which is common to all XSLT stylesheets.
- An undetermined quantity of `<xsl:template match=...>` instructions hangs from the root element.
- The value of match attribute for the first template that hangs off the root will be `“/”`. This template and its content never will be modified by the apply of operators. The only instruction inside this element will be `apply-templates`, that will have a select attribute whose value will be a `“/”` slash

---

<sup>4</sup>This company no longer exists, and the product seems to have been discontinued

followed by the root element name. Thus the rest of templates included in the stylesheet will be processed.

- The values for the match attributes for the rest of the templates from the second will be simply tag names of the input XML. Every value will have an undetermined number of children, that will be `apply-templates` or `value-of` instructions. These instructions will have select attributes, whose values will be XPath relative routes, built over the template path. Those routes would include every possible XPath clauses. `value-of` will be used instead of `apply-templates` if the when the value is self (`.`).

### 3.2 Second and thirds structure (types 2 and 3)

The main differences with the first one are:

- The value of the match attribute for the first template that hangs off the root will be `“/”` too, but, in this case it will have an indeterminate number of children, that will be all `apply-templates` instructions, whose values for the select attribute will be XPath absolute valid routes in the input XML, that will include only tag names separated to each other by a unique slash.
- The values for the match attributes for the other templates that hang from the XML root will be the same values that had the select attributes of the `apply-templates` in the first template. Therefore, there will be as many `template` instructions as the number of `apply-templates` in it, and they will be located in the same order.
- Every template of the previous section will have an undetermined number of children, and all of them will be `value-of` instructions, where the value for the select attribute will be XPath routes relative to the XPath absolute route of the father template. These routes would include every mechanisms of XPath that the designed operators allow.
- If the absolute route of a template has a maximum depth level inside the XML structure, its only `value-of` child will have select the self element: `“.”`.

Type 3 structure is identical to the previous one, but the children of the template instructions will be `apply-template` instead of `value-of` instructions, except when the XPath of the select attribute is `“.”`. This structure could be used when the two previous structures do not yield good results.

### 3.3 Genetic operators

The operators may be classified in two different types: the first one consists in operators that are commons to the three structures and whose assignment is to modify the XPath routes that contains the attributes of the XSLT instructions (specially `apply-template` and `value-of`). Operators in the second group are used

to modify the XSLT tree structure and take different shape in each of them (so that the structure is kept). In order to ensure the existence of the elements (tags) added to the XPath expressions and XSLT instruction attributes, every time one of them is needed it is randomly selected from the input file.

The common operators are:

- **XSLTreeMutatorXPath(Add|Mutate|Remove)Filter**: Adds, changes number, or removes a cardinal filter to any of the XPath tags that allow it. For example: `/book/chapter`  $\rightarrow$  `/book/chapter[4]`, `/book/chapter[2]`  $\rightarrow$  `/book/chapter[4]`, `/book/chapter[4]`, `/book/chapter[2]`  $\rightarrow$  `/book/chapter`.
- **XSLTreeMutatorXPathAddBranch**: Adds to a XPath route a new tag, chosen randomly from the possibles, observing the hierarchy of the input XML file tree: `/book/chapter`  $\rightarrow$  `/book/chapter/title`
- **XSLTreeMutatorXPathSetSelf**: Replaces the deepest node tag of a XPath route by the self node.
- **XSLTreeMutatorXPathSetDescendant**: Removes one of the intermediate tags from a XPath route, remaining a Descendant type node: `/book/chapter/title`  $\rightarrow$  `/book//title`.
- **XSLTreeMutatorXPathRemoveBranch**: Removes the deepest element tag of a XPath route, ascending a level in the XML tree. For example: `/book/chapter/title`  $\rightarrow$  `/book/chapter`.

The operators that change the DOM structure of the XSLT logicsheet are:

- **XSLTreeCrossoverTemplate**: Swaps template instructions subtrees between the two *parents*. This is the only crossover-like operator.
- **XSLTreeMutator(Add|Mutate|Remove)Template**: Inserts, changes or removes a template. Insertion is performed on the root element matching an random element. The choice of this random element gives more priority to the less deeper tags. The position of the new template inside the tree will be randomly selected, and its content will be `apply-templates` or `value-of` tags with the select attribute containing XPath routes relatives to the parent template XPath route randomly generated using the XPath operators. Change operates on a random node, generating a new subtree; and removal also eliminates a random template (if there are more than two).
- **XSLTree(Add|Remove)Apply**: It adds or removes a child to a randomly selected template present in the tree. The position of the new leaf inside the subtree that represents the template also will be randomly selected. The new element is randomly generated from the route that contains its parent template instruction. The **Remove** operator also deletes the template node if the removed child was the last remaining one, but it is not applied if there is a single template left.

- **XSLTreeMutateApply(1|2)**: Changes a randomly selected child (1) or creates a relative XPath from the one that contains the father XSLT:template and the XPath of the leaf that we are going to modify (2).
- **XSLTreeSetTemplateNull**: It chooses a subtree template from the XSLT tree and replaces its content by a single instruction `<xslt:value-of select=".">`. In the cases of second and third XSLT tree structure, there are nine equivalent operators.

### 3.4 Fitness function

Each XSLT sheet is applied to a XML input file and evaluated by comparing the result with the objective XML document. The evaluation function for each individual solution is represented next:

$$F = \frac{D}{L_1} + \left(\frac{S}{2}\right)^2 + \frac{L_2}{10000} \quad (1)$$

Where:

- $D$  represents the number of lines where the processed and objective XML documents differ.
- $L_1$  is the number of lines in the obtained XML.
- $L_2$  is the number of lines of the XSLT.
- $S$  corresponds to:
  - 0 when the number of lines in the objective XML -  $L_1 < 0$ .
  - $L_1$  - number of lines in the objective XML otherwise.

By taking into account not only the difference between the desired and obtained XML but also parameters related to the size of the resulting XSLT stylesheet; that way, there is a selective pressure for more compact programs, in an attempt to avoid bloating and useless structures.

## 4 Experiments and results

To test the algorithm we have performed several experiments with different XML input files and an unique XML output file. The algorithm has been executed five times for each input XML and for each of the three XSLT structures shown in the previous section.

The first input XML file (see figure 1) is a document that describes musical records from diverse authors, from which we want to extract the name of the authors and the first song of their disk, while the second XML input file includes one extra disk whose songs and author are not present in the XML output file.

```

<?xml version="1.0" encoding="ISO-8859-1"?>

<biblioteca_musical>
<disco>
  <titulo>I</titulo>
  <autor>Led Zeppelin</autor>
  <cancion>God Times, Bad Times</cancion>
  ...
  <fecha_grabacion>
    <mes>Mayo</mes>
    <mes>Junio</mes>
    <ano>1969</ano>
  ...
</biblioteca_musical>

```

Figure 1: Part of the first XML document used for experiments.

The computer used to perform the experiments is a Centrino Core Duo at 1.83 GHz, 2 GB RAM, and the Java Runtime Environment 1.6.0.01; the termination condition was set to 100 generations or until a solution was found and the selector was a Tournament selector with 5 individuals; 5 experiments were run, with different random seeds, for each template type and input document. The parser libraries used have been Xalan 2.7.0 for XSLT and Xerces 2.8.1 for XML. The parameters used in the experiments, and which were set to a default value with no attempt to optimize them, are shown in table 1; results are shown in tables 2 and 3.

These experiments assess the ability of the different evolution models to find a solution in a simple (document 1) and a slightly more complicated case (document 2). In the first case, it is not too difficult to find the solution in a few generations, but Type 2 templates are more successful than the rest, finding the solution in most cases, and doing so in less generations (thus, less evaluations); the XSLT logicsheet found is shown in figure 3. Type 3 is never able to find a solution in the given time. The second input document is more complex, and, in fact, 100 generations are not enough to find a solution; however, once again Type 1 and 2 are more successful, achieving an average minimum fitness of around 2.5 (optimum is close to 0, and is actually related to the minimum number of lines in the XSLT file divided by  $10^5$ ), and doing it in around 4 minutes; Type 3 needs half a minute more (on average) to reach the same number of generations, but results are worse than the other two types of templates; running time graphs are shown in figure 2.

## 5 Conclusions and Future Work

In this paper we present preliminary results of genetic programming applied to XSLT logicsheets, as opposed to Lisp S-Expressions or other type of programs;



Operator	Priority
XSLTTreeMutatorXPathSetSelf	0.10
XSLTTreeMutatorXPathSetDescendant	0.24 (Only Type 1)
XSLTTreeMutatorXPathRemoveBranch	0.27 (Type 2-3) 0.39 (Type 1)
XSLTTreeMutatorXPathAddBranch	0.99
XSLTTreeMutatorXPathAddFilter	0.45 (Type 2-3) 0.53 (Type 1)
XSLTTreeMutatorXPathMutateFilter	0.64 (Type 2-3) 0.69 (Type 1)
XSLTTreeMutatorXPathRemoveFilter	0.83
XSLTTreeCrossoverTemplate	0.61 (Type 1), 0.11 (Types 2 and 3)
XSLTTreeMutatorAddTemplate	0.13
XSLTTreeMutatorMutateTemplate	0.11
XSLTTreeMutatorRemoveTemplate	0.13
XSLTTreeAddApply	0.11
XSLTTreeMutateApply1	0.11
XSLTTreeMutateApply2	0.11
XSLTTreeRemoveApply	0.15
XSLTTreeSetTemplateNull	0.04

Table 1: Operator priorities (used for the roulette wheel that randomly selects the operator to apply) used in the experiments.

	Success rate	Time	Number of generations
Type 1	0.6	125968 $\pm$ 73509	66 $\pm$ 40
Type 2	0.8	19359 $\pm$ 9709	6 $\pm$ 4
Type 3	0	298528 $\pm$ 143834	100

Table 2: Results for the first input document; success rate is the number of times a solution is reached within the allotted 100 generations; running time is in milliseconds, and the number of generations needed to find the solution within the 100 allotted generations.

one of the advantages of this application is that resulting logicsheets can be used directly in a production environment, without the intervention of a human operator; besides, it tackles a real-world problem found in many organizations.

In these initial experiments we have found which kind of XSLT template structure is the most adequate for evolution, namely, one that matches the **select** attribute in **apply-templates** with the **match** attribute in templates, and an indeterminate number of value-of instructions within each template. By constraining evolution this way, we restrict the search space to a more reasonable size, and avoid the high degree of degeneracy of the problem, with many different structures yielding the same result, that, if combined, would result in invalid structures. In general, we have also proved that a XSLT logicsheet can be found just from an input/output pair of XML documents.

However, there are some questions and issues that will have to be addressed

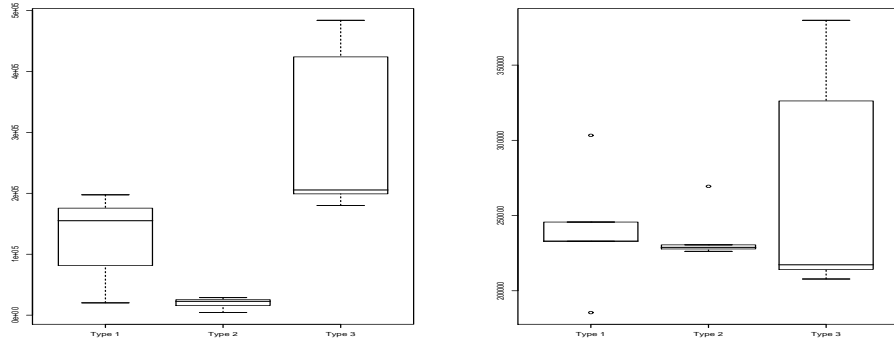


Figure 2: Distribution of running time (100 generations, or until a solution is found) for the first input document (left) and the second (right), in milliseconds. In the first case, Type 2 algorithm reaches the solution more frequently than the others, thus beating them in running time. However, in the second case, solution is not found during the 100 generations it was allowed to run; even so, Type 2 is consistently faster than the others.

	Fitness	Time
Type 1	$2.5 \pm 0.8$	$240069 \pm 42167$
Type 2	$2.4 \pm 1.1$	$236556 \pm 18430$
Type 3	10.77823	$269057 \pm 79014$

Table 3: Results for the second input document; in this case, success rate was 0 for all of them, so we show the average fitness of the best individual in the 100th generation, and running time.

in future papers:

- Using the DTD (associated to a XML file) as a source of information for conversions between XML documents and for restrictions of the possible variations.
- Adding different labels in the XSLT to allow the building of different kinds of documents such as HTML or WML.
- Considering the use of advanced XML document comparison tools (i.e. XMLdiff).
- Analyzing different XSLT processors. Current application uses Xalan but Saxon or XT might be faster.
- Testing evolution with other kind of tools, such as a chain of SAX filters.

```

<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output indent="no" method="xml"/>
  <xsl:template match="/">
    <biblioteca_musical>
      <xsl:apply-templates select="/biblioteca_musical/disco"/>
      <xsl:text>

      </xsl:text>
    </biblioteca_musical>
  </xsl:template>
  <xsl:template match="/biblioteca_musical/disco">
    <xsl:apply-templates select="autor"/>
    <xsl:text>

    </xsl:text>
    <xsl:apply-templates select="titulo"/>
    <xsl:text>

    </xsl:text>
  </xsl:template>
</xsl:stylesheet>

```

Figure 3: One of the Type 2 XSL logicsheets found as solution by the algorithm. In fact, this solution was found 4 out of the 5 times it was run.

- Obviously, testing different kinds and increasingly complex set of documents, and using several input and output documents at the same time, to test the generalization capability of the procedure.

## References

- [1] M. G. Arenas, B. Dolin, Juan-Julián Merelo-Guervós, P. A. Castillo, I. Fernández de Viana, and M. Schoenauer. JEO: Java Evolving Objects. In W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska., editors, *Poster Accepted at GECCO 2002*, page 991, 2002. Available from <http://geneura.ugr.es/pub/papers/MPP104.ps.gz>, <http://citeseer.nj.nec.com/context/2198210/0>.
- [2] M.G. Arenas, Pierre Collet, A.E. Eiben, Márk Jelasity, J. J. Merelo, Ben Paechter, Mike Preuß, and Marc Schoenauer. A framework for distributed evolutionary algorithms. Number 2439 in *Lecture Notes in Computer Science, LNCS*, pages 665–675. Springer-Verlag, September 2002. CiteSeer context: <http://citeseer.nj.nec.com/context/2189070/0>, available from

<http://link.springer.de/link/service/series/0558/papers/2439/243900665.pdf>,  
Metapress URL: <http://www.springerlink.com/link.asp?id=h4n29kbl69jvab4c>.

- [3] Z. Ben Miled, A. Farooq, M. Mahoui, N. Li, M. Dippold, and O. Bukhres. A wrapper induction application with knowledge base support: A use case for initiation and maintenance of wrappers. In *Proceedings - BIBE 2005: 5th IEEE Symposium on Bioinformatics and Bioengineering*, volume 2005, pages 65–72, 2005.
- [4] A. W. Biermann and G. Guiho, editors. *Computer Program Synthesis Methodologies*. Reidel, Dordrecht, 1983.
- [5] A.W. Biermann. The inference of regular LISP programs from examples. *IEEE Transactions on Systems, Man and Cybernetics*, 8(8):585–600, 1978.
- [6] James Clark. Xsl transformations (xslt) version 1.0 w3c recommendation 16 november 1999. Available from <http://www.w3.org/TR/xslt.html>.
- [7] James Clark and Steve DeRose. XML path language (XPath) version 1.0 w3c recommendation 16 november 1999. Available from <http://www.w3.org/TR/xpath>, November 1999.
- [8] Elliotte Rusty Harold. *XML Bible*. IDG Books worldwide, 1991.
- [9] E. Kuikka, P. Leinonen, and M. Penttonen. Towards automating of document structure transformations. In *Proceedings of the 2002 ACM Symposium on Document Engineering*, pages 103–110, 2002. Cited By (since 1996): 9.
- [10] P. Leinonen. Automating XML document structure transformations. In *Proceedings of the 2003 ACM Symposium on Document Engineering*, pages 26–28, 2003. Cited By (since 1996): 3.
- [11] Scott Martens. Automatic creation of XML document conversion scripts by genetic programming. In *Genetic Algorithms and Genetic Programming at Stanford*, page 269 ff., 2000.
- [12] Ute Schmid and J. Waltermann. Automatic synthesis of XSL-transformations from example documents. In M.H. Hamza, editor, *Artificial Intelligence and Applications Proceedings (IASTED International Conference on Artificial Intelligence and Applications (AIA 2004))*, pages 252–257, 2004.
- [13] Phillip D. Summers. A methodology for LISP program construction from examples. *J. ACM*, 24(1):161–175, 1977.
- [14] Wikipedia. Simple API for XML — Wikipedia, the free encyclopedia, 2007. [Online; accessed 21-March-2007].